Formal Method Mod. 2 (Model Checking)

Laboratory 7

Giuseppe Spallitta
giuseppe.spallitta@unitn.it

Università degli studi di Trento

April 26, 2022

Outline

History of nuXmv

SMV

Symbolic Model Verifier developed by McMillan in 1993.

NuSMV

Open-source symbolic model checker for SMV models. It has been developed by FBK, Carnegie Mellon University, University of Genoa and University of Trento.

nuXmv

Extends NuSMV for infinite state and timed (since v2) systems. Binary available for non-commercial or academic purposes only. Developed and maintained by the Embedded Systems unit of FBK.

Application of nuXvm

- nuXmv allows for the verification of:
  - *finite-state systems* through SAT and BDD based algorithms;
  - *infinite-state systems* (e.g. systems with *real* and *integer* variables) through SMT-based techniques running on top of **MathSAT5**;
  - *timed systems* (e.g. allows *clock* type) via reduction to infinite state model checking.
- nuXmv supports *synchronous* systems;
- *asynchronous* systems are no longer supported!

Outline

Interactive shell [1/3]

- `nuXmv -int` (or `NuSMV -int`) activates an interactive shell
- `help` shows the list of all commands (if a command name is given as argument, detailed information for that command will be provided).
  note: option `-h` prints the command line help for each command.
- `reset` resets the whole system (in order to read in another model and to perform verification on it).
- `read_model [-i filename]` sets the input model and reads it.
- `go`, `go_bmc`, `go_msat` initialize nuXmv for verification or simulation with a specific backend engine.

Interactive shell [2/3]

- `pick_state [-v] [-a] [-r | -i]` picks a state from the set of initial states.
  - `-v` prints the chosen state.
  - `-r` picks a state from the set of the initial states randomly.
  - `-i` picks a state from the set of the initial states interactively.
  - `-a` displays all state variables (requires `-i`).
- `simulate [-p | -v] [-a] [-r | -i] -k N` generates a sequence of at most `N` transitions starting from the `current state`.
  - `-p` prints the changing variables in the generated trace;
  - `-v` prints changed and unchanged variables in the generated trace;
  - `-a` prints all state variables (requires `-i`);
  - `-r` at every step picks the next state randomly.
  - `-i` at every step picks the next state interactively.
- `print_current_state [-h] [-v]` prints out the current state.
  - `-v` prints all the variables.

Interacting Shell [2/3] - Output Example

```
nuXmv > reset
nuXmv > read_model -i example01.smv ; go
nuXmv > pick_state -v; simulate -v
Trace Description: Simulation Trace
Trace Type: Simulation
  -> State: 1.1 <-
    b0 = FALSE
********  Simulation Starting From State 1.1   ********
Trace Description: Simulation Trace
Trace Type: Simulation
  -> State: 1.1 <-
    b0 = FALSE
  -> State: 1.2 <-
    b0 = TRUE
  -> State: 1.3 <-
    b0 = FALSE
  -> State: 1.4 <-
    b0 = TRUE
  -> State: 1.5 <-
    b0 = FALSE
  -> State: 1.6 <-
    b0 = TRUE
  ...
```

Interacting Shell [3/3]

- `show_vars [-s] [-f] [-i] [-t] [-v]` prints the variables content and type
    - `-s` print state variables;
    - `-f` print frozen variables;
    - `-i` print input variables;
    - `-t` prints the number of variables;
    - `-v` prints verbosely;
- `quit` stops the program.

Outline

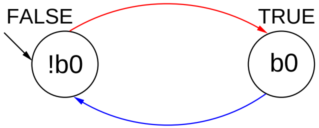
First SMV model

- an SMV model is composed by a number of **modules**;
- each **module** can contain:
  - state variable declarations;
  - formulae defining the valid *initial states*;
  - formulae defining the *transition relation*;

Example:

```
MODULE main
VAR
    b0 : boolean;

ASSIGN
    init(b0) := FALSE;
    next(b0) := !b0;
```



Basic Types [1/3]

boolean: TRUE, FALSE, ...

```
x : boolean;
```

enumerative:

```
s : {ready, busy, waiting, stopped};
```

bounded integers* (intervals):

```
n : 1..8;
```

*: integer numbers must be within C/C++ INT_MIN and INT_MAX bounds

Basic Types [2/3]

integers*: -1, 0, 1, …

```
  n : integer;
```

rationals: 1.66, f'2/3, 2e3, 10e-1, …

```
  r : real;
```

words: used to model arrays of bits supporting bitwise logical and arithmetic operations.

- `unsigned word[3];`
- `signed word[7];`

*: integer numbers must be within C/C++ INT_MIN and INT_MAX bounds

Basic Types [3/3]

arrays:
declared with a couple of lower/upper bounds for the index and a type

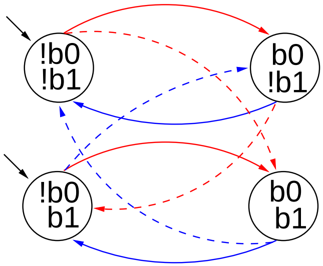
```
VAR
  -- array of 11 elements
  x : array 0..10 of boolean;
  -- array of 3 elements
  y : array -1..1 of {red, green, orange};
  -- array of array
  z : array 1..10 of array 1..5 of boolean;
ASSIGN
  init(x[5]) := bool(1);
  -- any value in the set
  init(y[0]) := {red, green};
  init(z[3][2]) := TRUE;
```

Array indexes *must be constants*;

Adding a state variable

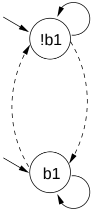
```
MODULE main
VAR
  b0 : boolean;
  b1 : boolean;

ASSIGN
  init(b0) := FALSE;
  next(b0) := !b0;
```



Remarks:

- the FSM is the result of the synchronous composition of the "subsystems" for b0 and b1
- the new state space is the cartesian product of the ranges of the variables.



Initial States [1/2]

Example:

```
init(x) := FALSE;      -- x must be FALSE
init(y) := {1, 2, 3}; -- y can be either 1, 2 or 3
```

`init(<variable>) := <simple_expression>;`

- constrains the initial value of `<variable>` to satisfy the `<simple_expression>`;
- the **initial** value of an unconstrained variable can be any of those allowed by its domain;

set of initial states

is given by the set of states whose variables satisfy all the `init()` constraints in a module.

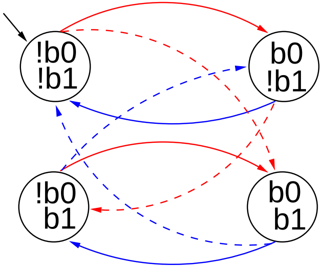
Initial States [2/2]

Example:

```
MODULE main
 VAR
   b0 : boolean;
   b1 : boolean;

 ASSIGN
   init(b0) := FALSE;
   next(b0) := !b0;

   init(b1) := FALSE;
```



Expressions [1/3]

- arithmetic operators:
  `+` `-` `*` `/` `mod` `-` (unary)
- comparison operators:
  `=` `!=` `>` `<` `<=` `>=`
- logic operators:
  `&` `|` `xor` `!` (not) `->` `<->`
- bitwise operators:
  `«` `»`
- set operators: `{v1,v2,...,vn}`
  - `in`: tests a value for membership in a set *(set inclusion)*
  - `union`: takes the union of 2 sets *(set union)*
- count operator: counts number of true *boolean* expressions
  `count(b1, b2, ..., bn)`

Expressions [2/3]

- case expression:

```
case
  c1   : e1;
  c2   : e2;
  ...
  TRUE : en;
esac
```

C/C++ equivalent:

```
if (c1) then e1;
else if (c2) then e2;
  ...
else en;
```

- if-then-else expression:

```
cond_expr ? basic_expr1 : basic_expr2
```

- conversion operators: `toint`, `bool`, `floor`, and
  - `swconst`, `uwconst`: convert an integer to a signed and an unsigned word respectively.
  - `word1` converts boolean to a single word bit.
  - `unsigned` and `signed` convert signed word to unsigned word and vice-versa.

Expressions [3/3]

- expressions in SMV do not necessarily evaluate to one value. In general, they can represent a set of possible values.

```
init(var) := {a,b,c} union {x,y,z};
```

- The meaning of `:=` in assignments is that the lhs can non-deterministically be assigned to any value in the set of values represented by the rhs.
- A constant `c` is considered as a syntactic abbreviation for `{c}` (the singleton containing `c`).

Transition Relation [1/2]

Transition Relation

specifies a constraint on the values that a variable can assume in the *next state*, given the value of variables in the *current state*.

```
next(<variable>) := <next_expression>;
```

- `<next_expression>` can depend both on "current" and "next" variables:

```
next(a) := { a, a+1 } ;
next(b) := b + (next(a) - a) ;
```

- `<next_expression>` must evaluate to values in the domain of `<variable>`;
- the **next** value of an unconstrained variable evolves non-deterministically;

Transition Relation [2/2]

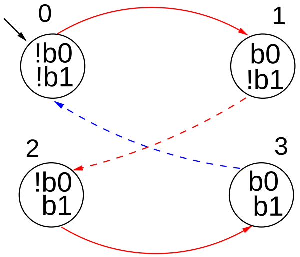
Example:

modulo-4 counter

```
MODULE main
 VAR
   b0 : boolean;
   b1 : boolean;

 ASSIGN
   init(b0) := FALSE;
   next(b0) := !b0;

   init(b1) := FALSE;
   next(b1) := case
       b0    : !b1;
       TRUE : b1;
     esac;
```



Output Variable [1/2]

output variable

A variable whose value deterministically depends on the value of other "current" state variables and for which no `init()` or `next()` are defined.

```
<variable> := <simple_expression>;
```

- `<simple_expression>` must evaluate to values in the domain of the `<variable>`.
- used to model *outputs* of a system;

Output Variable [2/2]

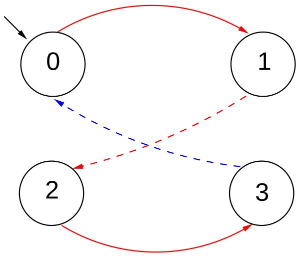
Example:

```
MODULE main
 VAR
   b0 : boolean;
   b1 : boolean;
   out : 0..3;

 ASSIGN
   init(b0) := FALSE;
   next(b0) := !b0;

   init(b1) := FALSE;
   next(b1) := ((!b0 & b1) | (b0 & !b1));

   out := toint(b0) + 2*toint(b1);
```



Assignment Rules (:=)

- **single assignment rule** – each variable may be assigned only once; Illegal examples:

```
init(var) := ready;
init(var) := busy;
```

```
var := ready;
var := busy;
```

```
next(var) := ready;
var := busy;
```

```
next(var) := ready;
next(var) := busy;
```

```
init(var) := ready;
var := busy;
```

Assignment Rules (:=)

- **single assignment rule** – each variable may be assigned only once; Illegal examples:

```
init(var) := ready;       var := ready;          next(var) := ready;
init(var) := busy;        var := busy;           var := busy;
-------------------       -------------------
next(var) := ready;       init(var) := ready;
next(var) := busy;        var := busy;
```

- **circular dependency rule** – a set of equations must not have “cycles” in its dependency graph, unless broken by delays; Illegal examples:

```
  xt(x) := next(y);     x := (x + 1) mod 2;     next(x) := x & next(x);
next(y) := next(x);
```

Legal example:

```
next(x) := next(y);
next(y) := y & x;
```

DEFINE declarations

```
DEFINE <id> := <simple_expression>;
```

- similar to *C/C++ macro* definitions: each occurrence of the defined symbol is replaced with the body of the definition
- provide an alternative way of defining *output variables*;

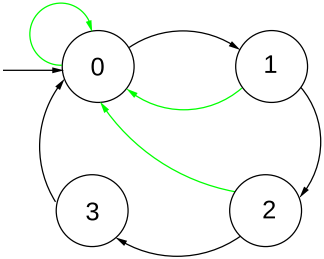
Example:

```
MODULE main
 VAR
   b0 : boolean;
   b1 : boolean;
 ASSIGN
   init(b0) := FALSE;
   next(b0) := !b0;
   init(b1) := FALSE;
   next(b1) := ((!b0 & b1) | (b0 & !b1));
 DEFINE
   out := toint(b0) + 2*toint(b1);
```

Example: modulo 4 counter with reset

The counter can be reset by an external "uncontrollable" signal.

```
MODULE main
VAR
  b0 : boolean; b1 : boolean; reset : boolean;
ASSIGN
  init(b0) := FALSE;
  init(b1) := FALSE;
  next(b0) := case
                reset = TRUE  : FALSE;
                reset = FALSE : !b0;
              esac;
  next(b1) := case
                reset : FALSE;
                TRUE  : ((!b0 & b1) | (b0 & !b1));
              esac;
DEFINE
  out := toint(b0) + 2*toint(b1);
```



Exercise 1

Exercise:
simulate the system with nuXmv and draw the FSM.

```
MODULE main
VAR
  request : boolean;
  state   : { ready, busy };

ASSIGN
  init(state) := ready;
  next(state) :=
    case
      state = ready & request : busy;
      TRUE                    : { ready, busy };
    esac;
```

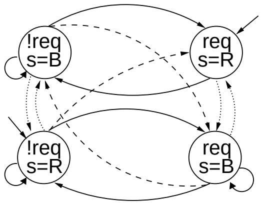
Exercise 1

Exercise:

simulate the system with nuXmv and draw the FSM.

```
MODULE main
VAR
  request : boolean;
  state   : { ready, busy };

ASSIGN
  init(state) := ready;
  next(state) :=
    case
      state = ready & request : busy;
      TRUE                    : { ready, busy };
    esac;
```



Constraint Style Modeling [1/4]

```
MODULE main
VAR
request : boolean;  state : {ready,busy};
ASSIGN
  init(state) := ready;
  next(state) := case
      state = ready & request : busy;
      TRUE                    : {ready,busy};
  esac;
```

Every program can be alternatively defined in a *constraint style*:

```
MODULE main
VAR
  request : boolean; state : {ready,busy};
INIT
  state = ready
TRANS
  (state = ready & request) -> next(state) = busy
```

Constraint Style Modeling [2/4]

- a model can be specified by zero or more constraints on:
  - *initial states*:
    `INIT <simple_expression>`
  - *transitions*:
    `TRANS <next_expression>`
  - *invariant states*:
    `INVAR <simple_expression>`
- constraints can be mixed with assignments;
- any propositional formula is allowed as constraint;
- not all **constraints** can be easily rewritten in terms of assignments!

```
TRANS
  next(b0) + 2*next(b1) + 4*next(b2) =
                          (b0 + 2*b1 + 4*b2 + tick) mod 8
```

Constraint Style Modeling [3/4]

assignment style
:

- by construction, there is always *at least one initial state*;
- by construction, all states have *at least one next state*;
- *non-determinism is apparent* (unassigned variables, set assignments...).

Constraint Style Modeling [4/4]

constraint style

:

- `INIT` constraints *can be inconsistent* $\Longrightarrow$ no initial state!
  - any specification (also `SPEC 0`) is vacuously true.
- `TRANS` constraints *can be inconsistent*: $\Longrightarrow$ deadlock state!

  Example:

  ```
  MODULE main
  VAR b : boolean;
  TRANS b -> FALSE;
  ```

  - **tip:** use `check_fsm` to detect deadlock states
- *non-determinism is hidden*:

  ```
  TRANS (state = ready & request) -> next(state) = busy
  ```

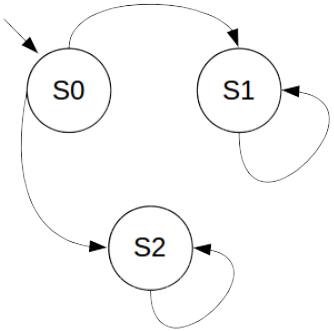
Example: Constraint Style & Case

```
MODULE main()
VAR
  state : {S0, S1, S2};

DEFINE
  go_s1 := state != S2;
  go_s2 := state != S1;

INIT
  state = S0;

TRANS
case
  go_s1 : next(state) = S1;
  go_s2 : next(state) = S2;
esac;
```



- **Q:** does it correspond to the FSM?

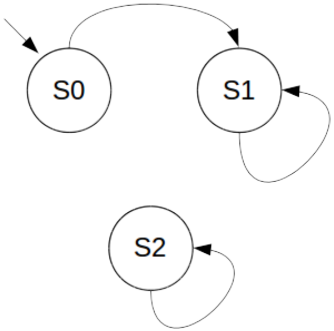
Example: Constraint Style & Case

```
MODULE main()
VAR
  state : {S0, S1, S2};

DEFINE
  go_s1 := state != S2;
  go_s2 := state != S1;

INIT
  state = S0;

TRANS
case
  go_s1 : next(state) = S1;
  go_s2 : next(state) = S2;
esac;
```



- Q: does it correspond to the FSM? No: cases are evaluated in order!

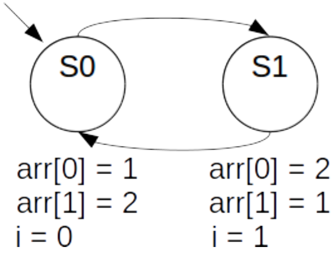
Example: Constraint Style & Swap

```
MODULE main()
VAR
   arr: array 0..1 of {1,2};
   i  : 0..1;

ASSIGN
  init(arr[0]) := 1;
  init(arr[1]) := 2;

  init(i) := 0;
  next(i) := 1-i;

TRANS
  next(arr[i]) = arr[1-i] &
  next(arr[1-i]) = arr[i];
```



- **Q:** does it correspond to the FSM?

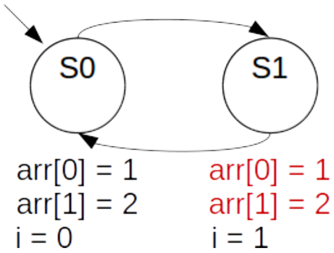
Example: Constraint Style & Swap

```
MODULE main()
VAR
   arr: array 0..1 of {1,2};
   i  : 0..1;

ASSIGN
  init(arr[0]) := 1;
  init(arr[1]) := 2;

  init(i) := 0;
  next(i) := 1-i;

TRANS
  next(arr[i]) = arr[1-i] &
  next(arr[1-i]) = arr[i];
```



- Q: does it correspond to the FSM? No: everything inside the **next()** operator is evaluated within the next state, indexes included!

Outline

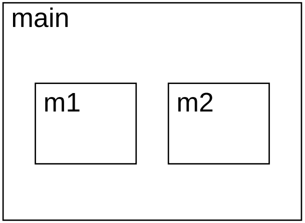
Modules [1/3]

SMV program = main module + 0 or *more* other modules

- a module can be instantiated as a VAR in other modules
- dot notation for accessing variables that are **local** to a module instance (e.g., m1.out, m2.out).

Example:

```
MODULE counter
  VAR out: 0..9;
  ASSIGN next(out) :=
                 (out + 1) mod 10;
MODULE main
  VAR m1 : counter; m2 : counter;
      sum: 0..18;
  ASSIGN sum := m1.out + m2.out;
```



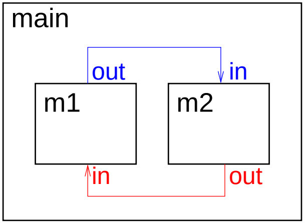
Modules [2/3]

A module declaration can be *parametric*:

- a parameter is passed by reference;
- any expression can be used as parameter;

Example:

```
MODULE counter(in)
  VAR out: 0..9;
  ...
MODULE main
  VAR m1 : counter(m2.out);
      m2 : counter(m1.out);
  ...
```



Modules [3/3]

- modules can be **composed**
- modules *without parameters and assignments* can be seen as simple **records**

Example:

```
MODULE point
VAR
  x: -10..10;
  y: -10..10;

MODULE circle
VAR
  center: point;
  radius: 0..10;
```

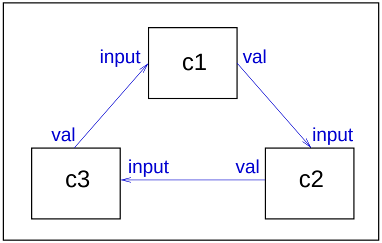
```
MODULE main
VAR c: circle;
ASSIGN
  init(c.center.x) := 0;
  init(c.center.y) := 0;
  init(c.radius)   := 5;
```

Synchronous composition [1/2]

The composition of modules is **synchronous** by default:
all modules move at each step.

```
MODULE cell(input)
VAR
  val : {red, green, blue};
ASSIGN
  next(val) := input;

MODULE main
VAR
  c1 : cell(c3.val);
  c2 : cell(c1.val);
  c3 : cell(c2.val);
```



Synchronous composition [2/2]

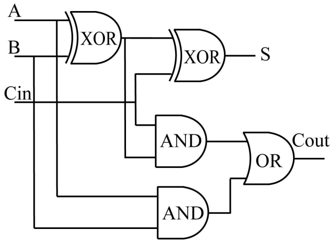
A possible execution:

step	*c1.val*	*c2.val*	*c3.val*
0	red	green	blue
1	blue	red	green
2	green	blue	red
3	red	green	blue
4	...	...	...
5	red	green	blue

Exercise: Adder [1/3]

Exercise 7.1: implementing adder

Implement a binary adder that takes into account two 4-bits numbers and returns their sum using an output variable. Implement both a bit-adder and the general adder as two separate modules.



Exercise: Adder [2/3]

```
MODULE bit-adder(in1, in2, cin)
VAR
  sum : boolean;
  cout : boolean;
ASSIGN
  next(sum) := (in1 xor in2) xor cin;
  next(cout) := (in1 & in2) | ((in1 xor in2) & cin);

MODULE adder(in1, in2)
VAR
  bit[0] : bit-adder(in1[0], in2[0], bool(0));
  bit[1] : bit-adder(in1[1], in2[1], bit[0].cout);
  bit[2] : bit-adder(in1[2], in2[2], bit[1].cout);
  bit[3] : bit-adder(in1[3], in2[3], bit[2].cout);
DEFINE
  sum[0] := bit[0].sum;
  sum[1] := bit[1].sum;
  sum[2] := bit[2].sum;
  sum[3] := bit[3].sum;
  overflow := bit[3].cout;
```

Exercise: Adder [3/3]

```
MODULE main
  VAR
    in1 : array 0..3 of boolean;
    in2 : array 0..3 of boolean;
    a : adder(in1, in2);

  ASSIGN
    next(in1[0]) := in1[0]; next(in1[1]) := in1[1];
    next(in1[2]) := in1[2]; next(in1[3]) := in1[3];
    next(in2[0]) := in2[0]; next(in2[1]) := in2[1];
    next(in2[2]) := in2[2]; next(in2[3]) := in2[3];
  DEFINE
    op1 := toint(in1[0]) + 2*toint(in1[1]) + 4*toint(in1[2]) +
         8*toint(in1[3]);
    op2 := toint(in2[0]) + 2*toint(in2[1]) + 4*toint(in2[2]) +
         8*toint(in2[3]);
    sum := toint(a.sum[0]) + 2*toint(a.sum[1]) + 4*toint(a.sum[2]) +
         8*toint(a.sum[3]) + 16*toint(a.overflow);
```

Outline

Homework

Homework 7.1: playing with Adder

- Simulate a random execution of the “adder” system;
- After how many steps the adder stores the computed final `sum` value? Is this number constant? Can you explain its behaviour?
- What happens if we initialize both sum and *cout* inside the bit-adder as FALSE? Can you explain which is the main difference with respect to the original algorithm?
- Can you modify the file in a simple way so that the sum is obtained after a single iteration? (PS: simple means you must modify/add less than 5 lines of code)
- Add a `reset` control which changes the values of the operands and restarts the computation of the sum

Homework

Homework 7.2: random calculator

Use nuXmv to create a "random" calculator: it creates two random arrays of 3 integers numbers in the range [1,10], then it randomly choose what operator apply for each pair of elements in the arrays (among sum, subtraction and multiplication) and store it in an output array of 3 elements called *res*. The results must be defined in 3 steps: in the first iteration you'll store the random operation between elements with index 0, in the second iteration the random operation between elements with index 1 and the same for the last index. Use an additional variable, *index*, to take into account this evolution.